

Batch-Expansion Training: An Efficient Optimization Paradigm for Machine Learning

Michał Dereziński

Dept. of Computer Science, University of California, Santa Cruz

MDEREZIN@UCSC.EDU

Dhruv Mahajan

Facebook Research, Menlo Park

DHRUVM@FB.COM

S. Sathiya Keerthi

Microsoft Corporation, Mountain View

KEERTHI@MICROSOFT.COM

S. V. N. Vishwanathan

Dept. of Computer Science, University of California, Santa Cruz

VISHY@UCSC.EDU

Markus Weimer

Microsoft Corporation, Redmond

MWEIMER@MICROSOFT.COM

Abstract

We propose Batch-Expansion Training (BET), a framework for running a batch optimizer on a gradually expanding dataset. As opposed to stochastic approaches, batches do not need to be resampled i.i.d. at every iteration, thus making BET more resource efficient in a distributed setting, and when disk-access is constrained. Moreover, BET can be easily paired with most batch optimizers, does not require any parameter-tuning, and compares favorably to existing stochastic and batch methods. We show that when the batch size grows exponentially with the number of outer iterations, BET achieves optimal $\tilde{O}(1/\epsilon)$ data-access convergence rate for strongly convex objectives.

1. Introduction

State-of-the-art optimization algorithms used in machine learning broadly tend to fall into two main categories: batch methods, which visit the entire dataset once before performing an expensive parameter update, and stochastic methods, which rely on a small subset of training data, to apply quick parameter updates at a much greater frequency. Both approaches present different trade-offs. Stochastic updates often provide very good early performance, since they can update the parameters a considerable number of times before even a single batch update finishes. On the other hand, by accessing the full dataset at each step,

batch algorithms can better utilize second-order information about the loss function, while taking advantage of parallel and distributed computing architectures. Finding approaches that provide the best of both worlds is an important area of research. In this paper, we propose a new framework which – while closer to batch in spirit – enjoys the benefits of stochastic methods. In addition, our framework addresses a practical issue observed in real-world industrial settings, which we now describe.

In industrial server farms, compute resources for large jobs become available only in a phased manner. When running a batch optimizer, which requires loading the entire dataset, one has to wait until all the machines become available before beginning the computation. Moreover, the training data, which typically consists of user logs, is distributed across multiple locations. This data needs to be normalized, often by communicating summary statistics or subsets of the data across the network. Since stochastic optimization algorithms only deal with a subset of the data at a time, one can largely avoid the bottleneck of data normalization by preparing mini-batches at a time. However, now each data point needs to be visited multiple times randomly, and this requires performing many random accesses from a hard-disk or network attached storage (NAS), which is inherently slow. Moreover, extending stochastic optimization to the distributed setting is still an active area of research. This raises the question of whether the compute and data-availability delays could be avoided in batch methods, and how that would affect the training time.

We propose Batch-Expansion Training (BET), a new op-

timization framework which addresses the above issues. Our framework hinges on the following observation: initially, when only a subset of the data is available, the statistical error (the error that arises because one is observing a sample from the true underlying data distribution) is large. Therefore, we can tolerate a large optimization error (the error that arises because of the iterative nature of the batch optimizer). However, as the sample size increases, the statistical error decreases and the corresponding optimization error that we can tolerate also decreases. BET exploits this by initially training models with large optimization error on smaller subsets of the data, and iteratively loading more training data, and driving down the optimization error. Relying on classical results in statistical learning theory, we show that for optimal performance, the data size should grow exponentially with the number of iterations. Moreover, we propose a simple and efficient algorithm, which can easily be paired with most batch optimizers, and does not require any parameter-tuning. Experiments using Nonlinear Conjugate Gradient (CG), as well as a Newton-CG method, demonstrate the versatility of our framework. We show that for strongly convex losses BET achieves the same asymptotic $\tilde{O}(1/\epsilon)$ convergence rate as SGD in terms of data accesses (and is strictly faster than regular batch updates). However, unlike stochastic methods, BET reuses all of the data that has already been loaded into memory, so when we take the cost of data-loading into account, experiments on multiple datasets show that our method outperforms stochastic as well as batch techniques.

2. Related Work

There has been a recent explosion of interest in optimization methods for machine learning, both in the batch and stochastic paradigms. Algorithms like Stochastic Variance Reduced Gradient method (Johnson & Zhang, 2013) and related approaches (Schmidt et al., 2013; Defazio et al., 2014) mix SGD-like steps with some batch computations to control the stochastic noise. Others have proposed to parallelize stochastic training through large mini-batches (Li et al., 2014; Dekel et al., 2012). However, these methods do not address the issues of compute and data-availability delays that we discussed above.

Additionally, two-stage approaches have been proposed (Agarwal et al., 2014; Shalev-Shwartz & Zhang, 2013), which employ SGD at the beginning followed by a batch optimizer (e.g., L-BFGS). These methods are much more limited than BET, which allows for multiple stages of optimization, with clear practical benefits.

Interleaving computation with data loading was shown to have significant practical benefits by Matsushima et al. (2012). However, that work is confined to training on a single machine and did not provide any theoretical conver-

gence guarantees. In contrast, we largely focus on the distributed setting and provide convergence guarantees.

The most relevant to our work is a stochastic variant of the Newton-CG method with increasing mini-batches (Byrd et al., 2012) as well as a similar approach proposed in (Friedlander & Schmidt, 2012). The algorithms proposed there still rely on stochastic sampling, which makes them less resource-efficient than BET, and not applicable to certain distributed settings (see Section 5 for a detailed comparison). The convergence guarantees offered in (Byrd et al., 2012) are limited to using gradient descent as the inner optimizer, and heavily rely on the independence conditions present in stochastic sampling. We use different proof techniques for time complexity analysis, which make the results applicable to a wider range of inner optimizers and do not require i.i.d. sampling.

3. Batch-Expansion Training

We consider a standard composite convex optimization problem arising in regularized linear prediction. Given a dataset $\mathbf{Z} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, we aim to approximately minimize the average regularized loss

$$\hat{f} \triangleq \frac{1}{N} \sum_{i=1}^N \ell_{z_i}(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|^2, \quad (1)$$

where $z_i = (\mathbf{x}_i, y_i)$ and $\ell_{z_i}(\mathbf{w}) \triangleq \ell(\langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle, y_i)$ is the loss of predicting with a linear model $\langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle$ against a target label y_i . Any iterative optimization algorithm in this setting will produce a sequence of models $\{\mathbf{w}_t\}_{t=1}^T$ with the goal that \mathbf{w}_T has small optimization error $\hat{g}(\mathbf{w}_T)$ with respect to the exact optimum $\hat{\mathbf{w}}^*$, where

$$\hat{g}(\mathbf{w}) \triangleq \hat{f}(\mathbf{w}) - \hat{f}(\hat{\mathbf{w}}^*) \quad \text{and} \quad \hat{\mathbf{w}}^* \triangleq \arg \min_{\mathbf{w}} \hat{f}(\mathbf{w}).$$

Note that our true goal is to predict well on an unseen example $z = (\mathbf{x}, y)$ coming from an underlying distribution. The right regularization λ for this task can be determined experimentally or from the statistical guarantees of loss function ℓ , as discussed in (Sridharan et al., 2009; Shalev-Shwartz & Srebro, 2008). In this paper, however, we will assume that an acceptable λ was chosen and concentrate on the problem of minimizing \hat{f} .

3.1. Linear convergence of the batch optimizer

Adding 2-norm regularization in function \hat{f} makes it λ -strongly convex. In this setting, many popular batch optimization algorithms enjoy linear convergence rate (Nocedal & Wright, 2006). Namely, given arbitrary model \mathbf{w} and any $c > 1$, after at most $\mathcal{O}(\log(c))$ iterations - where a single iteration can look at the entire dataset - we can reduce its optimization error by a multiplicative factor of c ,

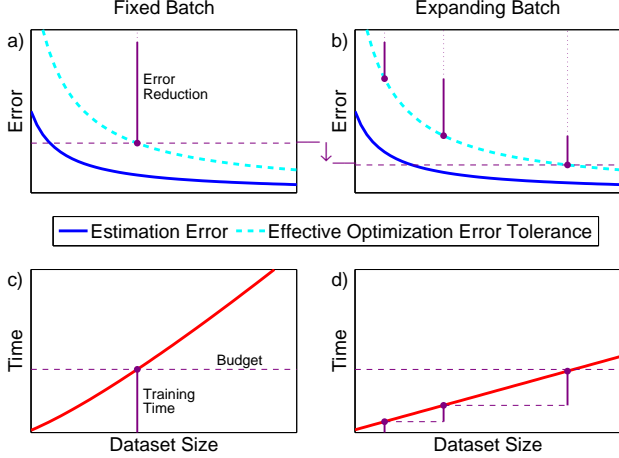


Figure 1. Estimation error and effective optimization error tolerance are inversely proportional to the dataset size (plots a and b). In Fixed Batch we preselect a data size and then reduce the optimization error (vertical line in a), whereas in Expanding Batch we divide the work into stages with different data sizes (vertical lines in b). Optimization is faster for smaller data sizes (see plots c and d), so Expanding Batch reaches smaller effective optimization error tolerance within the same time budget compared to Fixed Batch (plots a and b).

obtaining \mathbf{w}' such that

$$\hat{g}(\mathbf{w}') \leq c^{-1} \cdot \hat{g}(\mathbf{w}).$$

Note, that the runtime of a single iteration will depend on data size N , but the number of needed iterations does not.

Setting $c = 2$, we observe that only a constant number of batch iterations is necessary for a linearly converging optimizer to halve the optimization error of \mathbf{w} . This constant depends on the convergence rate enjoyed by the method. For example, in the case of batch gradient descent we need $\mathcal{O}(1/\lambda)$ iterations to reduce the optimization error by a factor of 2 (Bubeck, 2015) (note the dependence on the strong convexity coefficient), while other methods (like L-BFGS) can achieve better rates of convergence (Lin et al., 2007). Furthermore, the time complexity of performing a single iteration for many of those algorithms (including GD and L-BFGS) is linearly proportional to the data size. We refer to methods exhibiting both linear convergence (with respect to a given loss) and linear time complexity of a single iteration as *linear optimizers*. From now on, we only consider this class of methods.

3.2. Dataset size selection

The general task of an optimization algorithm is to return a model with small optimization error $\hat{g}(\mathbf{w}) \leq \epsilon$. For selecting an effective optimization error tolerance ϵ in a machine

learning problem, we often look at the estimation error exhibited by the objective, i.e. how much \hat{f} deviates from the expected regularized loss measured on a random unseen example. As discussed in (Shalev-Shwartz & Srebro, 2008), the effective optimization error tolerance should be proportional to the estimation error of \hat{f} , because optimizing beyond that point does not yield any improvement on unseen data. Note that under standard statistical assumptions, the more data we use, the smaller the estimation error becomes (Sridharan et al., 2009), and thus, the effective optimization error tolerance should also decrease (see Figure 1a).

Consider the scenario where data is abundant, but we have a limited time budget for training the model¹. In this case, a practitioner would select the data size so that we can reach the smallest effective optimization error tolerance (dashed curve in Figure 1a) within the allotted training time. Vertical line in Figure 1a shows the reduction in optimization error that the algorithm has to achieve to obtain the desired tolerance. If we use a batch optimizer for this task, the training time is determined by two factors. First, as the dataset grows, each iteration takes longer (e.g. for linear optimizers the iteration time is proportional to the data size, as discussed in Section 3.1). Second, the algorithm has to perform larger number of iterations, the smaller the effective optimization error tolerance. Combined, those two effects result in training time, which, for linear optimizers, grows faster than linearly with the data size (see Figure 1c).

In this paper, we propose that instead of finding the optimal dataset size at the beginning, we first load a small subset of the data, train the model until we reach the corresponding effective optimization error tolerance, then we load more data, and optimize further, etc. Vertical lines in Figure 1b illustrate how the optimization error is reduced in the multiple stages working with increasing dataset sizes. This procedure benefits from inexpensive iterations in the early stages, as shown in Figure 1d, with vertical lines corresponding to the training time for each data size, and the horizontal line showing the time budget (note that the budget shown for this method is the same as the one used in Figure 1c). Thus, our approach is able to reach smaller effective optimization error tolerance within the same time budget compared to fixing the dataset size at the beginning, as seen by comparing Figures 1a and 1b.

3.3. Exponentially increasing batches

We now precisely formulate the idea of training with gradually increasing data size. Suppose our goal is to return a model $\hat{\mathbf{w}}$ with optimization error $\hat{g}(\hat{\mathbf{w}}) \leq \epsilon$. In the procedure described above, we seek to obtain a sequence of gradually improving models $\mathbf{w}_1, \dots, \mathbf{w}_{T-1}$ before we reach

¹This is a reasonable implementation practice in many web applications.

$\mathbf{w}_T = \hat{\mathbf{w}}$. Any of the intermediate models (say, model \mathbf{w}_t for $t < T$) has a large optimization error relative to $\hat{\mathbf{w}}$, so to compute \mathbf{w}_t we can minimize an objective function with correspondingly large estimation error. Thus, we will first obtain \mathbf{w}_1 with optimization error² ϵ_1 using n_1 data points, next we pick a smaller error tolerance $\epsilon_2 < \epsilon_1$ and bigger data size $n_2 > n_1$, computing a better model \mathbf{w}_2 , etc. so that in the end we reach $\epsilon_T = \epsilon$. Algorithm 1 demonstrates a simple instantiation of this strategy, where at each stage we double the data size:

Algorithm 1 Batch-Expansion Training

```

Pick initial model  $\mathbf{w}_0$ 
Load first  $n_1$  data points
for  $t = 1..T$  do
     $\mathbf{w}_t \leftarrow$  run  $\kappa_t$  iterations on all  $n_t$  loaded data points
     $n_{t+1} \leftarrow b_t n_t$     increase data size,  $b_t > 1$ 
    Load additional  $n_{t+1} - n_t$  data points
end for
return  $\mathbf{w}_T$ 
    
```

The basis for the number of inner iterations κ_t will be explained below. Note that at stage t of the process we are working with an estimate of the loss function

$$\hat{f}_t(\mathbf{w}) \triangleq \frac{1}{n_t} \sum_{i=1}^{n_t} \ell_{z_i}(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|^2,$$

which tends to \hat{f} with increasing t . At this stage, the optimizer is in fact converging to an approximate optimum

$$\hat{\mathbf{w}}_t^* \triangleq \arg \min_{\mathbf{w}} \hat{f}_t(\mathbf{w}),$$

rather than to the minimizer of \hat{f} , $\hat{\mathbf{w}}^*$. To decide how much data is needed at each stage we will describe the relationship between data size n_t and the desired optimization error ϵ_t . Note, that model \mathbf{w}_t obtained at stage t is assumed to satisfy $\hat{g}_t(\mathbf{w}_t) \leq \epsilon_t$, where \hat{g}_t represents optimization error for the given data subset, i.e. $\hat{g}_t(\mathbf{w}) = \hat{f}_t(\mathbf{w}) - \hat{f}_t(\hat{\mathbf{w}}_t^*)$. When going to the next stage, the data size increases, and thus we can use optimization error function \hat{g}_{t+1} , which is a better estimate of \hat{g} , than \hat{g}_t is. In Appendix B.2 of (Auteurs, 2017), we show that function \hat{g}_{t+1} can be uniformly bounded by \hat{g}_t , plus an additional term which can be interpreted as the estimation error (of \hat{g}_t with respect to \hat{g}_{t+1}):

$$\hat{g}_{t+1}(\mathbf{w}) \leq 2 \cdot \hat{g}_t(\mathbf{w}) + \mathcal{O}(1/(\lambda n_t)).$$

As discussed earlier, the effective optimization error for data size n_t is proportional to the estimation error suffered by \hat{g}_t , hence it is sufficient to demand that

$$\hat{g}_t(\mathbf{w}_t) \leq \epsilon_t \triangleq \mathcal{O}(1/(\lambda n_t)). \quad (2)$$

²The optimization error for intermediate models \mathbf{w}_t is computed using only the loaded portion of the data.

Note that this makes the optimization error tolerance ϵ_t inversely proportional to the subset size n_t , confirming our intuition that for $t < T$, since $\epsilon_t > \epsilon$, we can work with a batch of size n_t that is smaller than N .

To establish the correct rate of growth for the data size n_t , we combine (2) with the observations made in Section 3.1. First, note that the rate of decay of sequence $\{\epsilon_t\}$ should match the convergence rate of the optimization algorithm, so that inequality $\hat{g}_t(\mathbf{w}_t) \leq \epsilon_t$ can be satisfied for all t . Recall from Section 3.1, that a linear optimizer takes only a constant number of iterations (call it κ) to reduce the optimization error by a factor of 2. This suggests the following simple strategy: at each stage perform κ iterations of the optimizer, then divide the tolerance level by 2 (matching the convergence rate). Note that based on Equation (2), this corresponds to doubling the data size n_t . Thus, we obtain the following simple scheme for data expansion, which maintains the desired relationship between n_t and ϵ_t :

$$\epsilon_{t+1} = \epsilon_t/2, \quad n_{t+1} = 2 \cdot n_t.$$

It is important that n_t grows exponentially with t . This allows for a considerable improvement in runtime. Let us say that $\epsilon_0 = \mathcal{O}(1)$ and T stages are needed to reach the final desired tolerance ϵ , i.e. $T = \log(\epsilon_0/\epsilon) = \mathcal{O}(\log(1/\epsilon))$. Moreover, using (2) the suitable size of the full dataset is $N = \mathcal{O}(1/(\lambda\epsilon))$. If we assume that one iteration of the linear optimizer takes time proportional to the data size, then the time complexity of the optimization when using batch-expansion is given by

$$\sum_{t=1}^T \kappa n_t = \kappa n_0 \sum_{t=1}^T 2^t = \mathcal{O}(\kappa N) = \mathcal{O}\left(\frac{\kappa}{\lambda\epsilon}\right).$$

On the other hand, when running the same optimizer on full dataset from the beginning, the time complexity becomes

$$\sum_{t=1}^T \kappa N = \kappa N \cdot T = \mathcal{O}\left(\frac{\kappa}{\lambda\epsilon} \cdot \log(1/\epsilon)\right).$$

Note that to establish convergence of the proposed algorithm, it is only required that the dataset is randomly permuted, i.e. that each subset $\{z_i\}_{i=1}^{n_t}$ represents a random portion of the data. However, the batches used in different stages do not need to be independent of each other, which is why we can reuse data from previous stages. Section 4 precisely formulates the ideas discussed above. Moreover, a careful analysis of the time complexity for this approach is given in Theorem 4.1, showing that the method indeed exhibits $\tilde{\mathcal{O}}(\kappa/(\lambda\epsilon))$ convergence rate for optimal values of its input parameters. Next, we will discuss a parameter-free approach, which works well in practice.

3.4. Two-track algorithm

How many iterations of the proposed expansion procedure should be performed at each stage? From our high-level analysis, we concluded that a roughly constant number of updates should be sufficient for any stage (using a linear optimizer), since each time we aim to make the same multiplicative improvement to the optimization error suffered by our model. However, that constant may depend on the type of loss function, the dataset, as well as the optimizer used, and moreover, in practice the right number of iterations may in fact vary to some extent between the stages. Therefore, we need a practical method of deciding the right time to double the data. Consider the following experiment: we run two optimization tracks in parallel, first one for the batch of size n_t , the other for half of that batch. For one slower step on the bigger batch, two faster steps are performed on the smaller one. Which track will make better progress towards the optimum of the bigger batch? If the starting model is far enough from the optimum $\hat{\mathbf{w}}_t^*$, then the faster updates will initially have an advantage. However, as the convergence proceeds, only the slower track can get arbitrarily close, so at some point it will move ahead of the fast one (in terms of the loss $\hat{f}_t(\mathbf{w})$). Denote the starting model as $\mathbf{w}_{t,0}$. The secondary track (running on half of the batch) also starts at the same point, denoted as $\mathbf{w}'_{t-1,0} = \mathbf{w}_{t,0}$, and they are both updated as follows:

$$\begin{aligned}\mathbf{w}_{t,s+1} &\leftarrow \text{Update}(\mathbf{w}_{t,s}, n_t), \\ \mathbf{w}'_{t-1,s+1} &\leftarrow \text{Update}(\mathbf{w}'_{t-1,s}, n_{t-1}),\end{aligned}$$

where $\text{Update}(\mathbf{w}, n)$ is one step of the optimizer, with respect to model \mathbf{w} , on the batch $\{z_i\}_{i=1}^n$. To reflect faster update time for the second track, we let it make twice as many updates when comparing to the first track. Thus, after s iterations of the above procedure, we can use the following easily testable condition for when the first track becomes better than the second one:

$$\hat{f}_t(\mathbf{w}_{t,\lfloor s/2 \rfloor}) < \hat{f}_t(\mathbf{w}'_{t-1,s}). \quad (3)$$

Algorithm 2 describes Batch-Expansion Training implemented using the Two-Track strategy. Note that in the algorithm for each step of the first track, we run one step of the second track (as opposed to 2 steps), which reduces the amount of extra computation per iteration, that is used to evaluate Condition (3).

3.5. Discussion

The choice to increase data size by a factor of 2 at each stage (rather than by a different factor) is not crucial for the optimization performance (both theoretically and in practice), therefore this parameter does not require tuning. The initial subset size n_0 also does not affect performance sig-

Algorithm 2 Two-Track Optimizer

```
Initialize  $\mathbf{w}_{1,0} = \mathbf{w}'_{0,0}$  arbitrarily
Pick any  $2 \leq n_1 = 2n_0 < N$ 
Initialize  $s \leftarrow 0$  and  $t \leftarrow 1$ 
while  $n_t < N$  do
   $\mathbf{w}_{t,s+1} \leftarrow \text{Update}(\mathbf{w}_{t,s}, n_t)$ 
   $\mathbf{w}'_{t-1,s+1} \leftarrow \text{Update}(\mathbf{w}'_{t-1,s}, n_{t-1})$ 
   $s \leftarrow s + 1$ 
  if  $\hat{f}_t(\mathbf{w}_{t,\lfloor s/2 \rfloor}) < \hat{f}_t(\mathbf{w}'_{t-1,s})$  then
     $n_{t+1} \leftarrow 2n_t$ 
     $\mathbf{w}'_{t,0}, \mathbf{w}_{t+1,0} \leftarrow \mathbf{w}_{t,s}$ 
     $t \leftarrow t + 1$ 
     $s \leftarrow 0$ 
  end if
end while
while stopping condition not met do
   $\mathbf{w}_{t,s+1} \leftarrow \text{Update}(\mathbf{w}_{t,s}, N)$ 
   $s \leftarrow s + 1$ 
end while
return  $\mathbf{w}_{t,s}$ 
```

nificantly - generally, the larger n_0 we select, the more updates will be performed before first data expansion, but as long as the initial subset is small enough, total optimization time will remain close to optimal. Thus, Algorithm 2 does not require any tuning to achieve good performance. Moreover, our method can be paired with many popular batch optimizers, and it will automatically adapt its behavior to the selected inner optimizer, as shown in Appendix A.1 of (Authors, 2017).

It is important to note that the fraction of data accessed by the algorithm is only gradually expanded as optimization proceeds. Moreover, BET iterates multiple times over the data points that have already been loaded. Thus, it is very resource efficient in a way that can be beneficial with:

Slow disk-access. Loading data from disk to memory can be a significant bottleneck (Matsushima et al., 2012). Performing multiple iterations over the data points while extra data is being loaded in parallel provides speed-up.

Resource ramp-up. In distributed computing, often not all resources are made available immediately at the beginning of the optimization (Narayanamurthy et al., 2013), which similarly leads to gradual data availability.

4. Complexity Analysis

In this section, we provide theoretical guarantees for the time complexity of Batch-Expansion Training and compare them to other approaches. For the remainder of this section, we assume that the inner optimizer for some $\kappa > 1$ and for

every t , \mathbf{w} exhibits linear convergence:

$$\hat{g}_t(\text{Update}(\mathbf{w}, n_t)) \leq (1 - (1/\kappa)) \hat{g}_t(\mathbf{w}).$$

4.1. Data-Access Complexity

For the sake of complexity analysis, we discuss a parameterized variant of our approach, described in Algorithm 3, and establish complexity results for it. Here, the number of updates needed at each stage is a fixed parameter $\hat{\kappa}$.

Algorithm 3 Optimal BET

Input: Target tolerance ϵ
 Pick ϵ_0, n_0
 Initialize $\mathbf{w}_0 \leftarrow 0$, $t \leftarrow 0$ and $\hat{\kappa} = \lceil \kappa \log(6) \rceil$
while $3 \cdot \epsilon_t > \epsilon$ **do**
 $n_{t+1} \leftarrow 2n_t$
 $\mathbf{w}_{t,0} \leftarrow \mathbf{w}_t$
 for $s = 1..\hat{\kappa}$ **do**
 $\mathbf{w}_{t,s} \leftarrow \text{Update}(\mathbf{w}_{t,s-1}, n_{t+1})$
 end for
 $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_{t,\hat{\kappa}}$
 $\epsilon_{t+1} \leftarrow \epsilon_t/2$
 $t \leftarrow t + 1$
end while
 $T \leftarrow t$
return \mathbf{w}_T

Assumptions. We assume that the feature mapping $\phi(\cdot)$ (see (1) and the discussion below it) is B -bounded, i.e. it satisfies $\|\phi(\mathbf{x}_i)\| \leq B$, and that the loss function $\ell_{z_i}(\mathbf{w}) = \ell(z_i, \mathbf{w})$ is L -Lipschitz in z_i for all \mathbf{w} . Moreover, we will use the fact that \hat{f} is λ -strongly convex, due to the use of 2-norm regularization. The result below (proven in Appendix B of (Authors, 2017)) holds with high probability with respect to a random permutation of data.

Theorem 4.1 *For any n_0, δ, ϵ there exists ϵ_0 such that data-access complexity of Algorithm 3 is*

$$\mathcal{O}\left(\frac{\kappa}{\lambda\epsilon} \cdot (\log \log(1/\epsilon) + \log(1/\delta))\right)$$

and w.p. at least $1 - \delta$ we have $\hat{f}(\mathbf{w}_T) - \hat{f}(\hat{\mathbf{w}}^*) \leq \epsilon$.

Using gradient descent as the inner optimizer, we have $\kappa = \mathcal{O}(1/\lambda)$, so Algorithm 3 reaches data-access complexity $\tilde{\mathcal{O}}(1/(\lambda^2\epsilon))$. However, the general nature of this approach allows us to choose a different linear optimizer with better guarantees, like $\kappa = \mathcal{O}(1/\sqrt{\lambda})$ for accelerated gradient descent. Methods like L-BFGS and other approximate Newton algorithms have been shown to exhibit linear convergence (Lin et al., 2007; Erdogdu & Montanari, 2015) with a rate that does not suffer from such strict dependence on the strong convexity coefficient λ . Hence, when using

those optimizers, for most problems we should expect κ to be a small constant factor, in which case data-access complexity becomes $\tilde{\mathcal{O}}(1/(\lambda\epsilon))$.

4.2. Time Complexity

We will now discuss a simple model of time complexity for BET, which incorporates such aspects as hardware-acceleration and data-loading. Our aim is to highlight certain regimes in the computational architecture landscape which allow for a meaningful comparison of stochastic and batch methods. We compare BET with three other paradigms for running an inner batch optimizer update:

1. **Batch:** using the inner optimizer as a regular batch method on the entire dataset.
2. **Dynamic Sample Method (DSM):** applying stochastic sampling with dynamically increasing sample sizes, proposed in (Byrd et al., 2012),
3. **Mini-Batch:** applying stochastic sampling with mini-batches of fixed size b (Li et al., 2014).

First, we compare data-access complexity of the methods with their sample complexity $\mathcal{N}_*(\epsilon)$ (number of unique data points needed to reach generalization error ϵ). For stochastic methods, sample complexity essentially corresponds to their data-access complexity. On the other hand, for batch methods time complexity depends both on the sample size $\mathcal{N}_*(\epsilon)$ and separately on ϵ itself (for example to incorporate linear convergence rate $\kappa \log(1/\epsilon)$). For Batch-Expansion Training, data-access complexity also does not equal sample complexity, but the gap does not depend on ϵ .

All of the considered methods exhibit sample complexity $\tilde{\mathcal{O}}(1/(\lambda\epsilon))$, however with considerably different constant factors in the stochastic and batch settings, which makes a direct comparison more challenging. To that end, we define multiplicative factors for DSM and Mini-Batch³:

$$\text{DSM: } \kappa_d \triangleq \mathcal{N}_{\text{DSM}}(\epsilon)/\mathcal{N}_{\text{BET}}(\epsilon), \quad (4)$$

$$\text{Mini-Batch: } \kappa_m \triangleq \mathcal{N}_{\text{MB}}(\epsilon)/\mathcal{N}_{\text{BET}}(\epsilon). \quad (5)$$

To derive time complexity $\mathcal{T}_*(\epsilon)$ for the methods, we make the following assumptions on the computing architecture:

1. Processing one data point takes $1/p$ units of time, where p is a hardware-acceleration factor.
2. Data points are loaded sequentially, as the optimization progresses, one point per every a units of time.
3. We incur an overhead of s units in between each two consecutive calls to the inner optimizer.

Note, that this model is by no means exhaustive, for example it does not take into account aspects related to the dimensionality of the data. We can think of this dimension-

³For BET and Batch, we use the same factor κ , which corresponds to the convergence rate of the inner optimizer

Method	$\mathcal{T}_*(\epsilon)/\mathcal{N}_{\text{BET}}(\epsilon)$
Batch	$\mathcal{O}\left(a + \frac{\kappa \log(1/\epsilon)}{p}\right)$
BET	$\mathcal{O}\left(a + \frac{\kappa}{p}\right)$
DSM	$\mathcal{O}\left(\left(a + \frac{1}{p}\right) \cdot \kappa_d\right)$
Mini-Batch	$\mathcal{O}\left(\left(a + \frac{1}{p} + \frac{s}{b}\right) \cdot \kappa_m\right)$

Table 1. Comparison of time complexities for methods.

dependence as being absorbed into the time units used in the analysis, which is reasonable as we are only considering optimizers with linear time dependence on the dimension.

Table 1 compares time complexities under the proposed model, normalized by the BET sample complexity $\mathcal{N}_{\text{BET}}(\epsilon)$ (thus, the term $1/(\lambda\epsilon)$ cancels out from the formulas). Unlike for BET, the available theoretical guarantees for DSM and Mini-Batch apply only to particular inner optimizers (e.g. gradient descent). Note that Mini-Batch time complexity is based on the convergence rate of the EMSO method (Li et al., 2014), (which is better than the rate of simple mini-batch SGD for large b). In this big-picture analysis, for the sake of comparison we extrapolated those results to cover all inner optimizers exhibiting linear convergence (the convergence rate gets absorbed into the factor κ_*). Note, that some bounds for stochastic methods carry an additional $\log(1/\epsilon)$ term, however there are results for SGD, which avoid that term under additional smoothness assumptions (Bottou & Bousquet, 2007), so for clarity of presentation we elected to leave it out of this comparison. Factors κ , κ_d and κ_m depend mainly on the inner optimizer. In our experiments, their values were comparable for all methods, ranging between 2 and 4.

The formulas in Table 1 can essentially be divided into three components: data-availability, computation time and sequentiality cost. The last one is only present for Mini-Batch, where it can dominate the optimization time, especially when mini-batch size b is small. For other methods, this cost is negligible for small enough ϵ . Moreover, note that by the very nature of non-stochastic approaches (Batch and BET), their dependence on data-availability is only limited by sample complexity of the learning problem, hence there are no additional multiplicative factors there. In practice, data-loading can happen concurrently to the computations. BET is especially well suited in that case, since it allows the κ/p term to be absorbed by a .

As for the computation time, each method carries a κ_*/p term, with the exception of Batch, which suffers an additional $\log(1/\epsilon)$ factor, clearly observed experimentally in the comparison between Batch and BET. As we can see, the preferred optimization method depends on the parameters of the architecture as well as the learning problem, but

Dataset	Instances (train / test)	Features
w8a	49,749 / 14,951	300
rcv1	10,121 / 10,121	47,236
real-sim	36,154 / 36,155	20,958
webspam	175,000 / 175,000	16,609,143
SUSY	4,096,000 / 500,000	18

Table 2. A list of datasets used for the experiments.

we can reach a few general conclusions:

1. BET is asymptotically superior to Batch for any parametrization of the time complexity model.
2. If data availability is slow relative to numerical computations and $\kappa_d > 1$, then BET is faster than DSM.
3. Mini-Batch is slower than other methods when the sequentiality cost is high.

5. Experiments

In this section we present experimental results supporting the theoretical time complexity analysis from Section 4. As the optimization problem we use squared hinge-loss for SVM with 2-norm regularization trained on several standard LIBSVM datasets (see Table 2). The regularization parameter for each dataset was selected by tuning to achieve highest test-set accuracy. All algorithms start with the initial model vector \mathbf{w} set to all zeros. BET was implemented as shown in Algorithm 2. The initial subset size n_0 was tested in the range between 100 and 2000 data-points, with minimal effect on performance. As the main inner optimizer for BET we used Sub-sampled Newton-CG (SN) (Byrd et al., 2011). Additional experiments in Appendix A.1 of (Authors, 2017) show how our algorithm performs when paired with nonlinear Conjugate Gradient (Fletcher & Reeves, 1964). We compare BET against:

1. Fixed Batch strategy using SN as the optimizer,
2. Dynamic Sample Method (DSM) (Byrd et al., 2012),
3. Adagrad (Duchi et al., 2011).

Both DSM and Adagrad were tuned to obtain optimal parameter values for each algorithm. Performance of DSM varied considerably depending on its parameters, as discussed in Appendix A.2 of (Authors, 2017). BET does not require any tuning.

5.1. Optimization Time

In this section, we evaluate algorithms using time complexity model described in Section 4.2. We present the results in terms of log Relative Functional Value Difference

$$\log \text{RFVD: } \log \left((\hat{f}(\mathbf{w}) - \hat{f}(\hat{\mathbf{w}}^*)) / \hat{f}(\hat{\mathbf{w}}^*) \right). \quad (6)$$

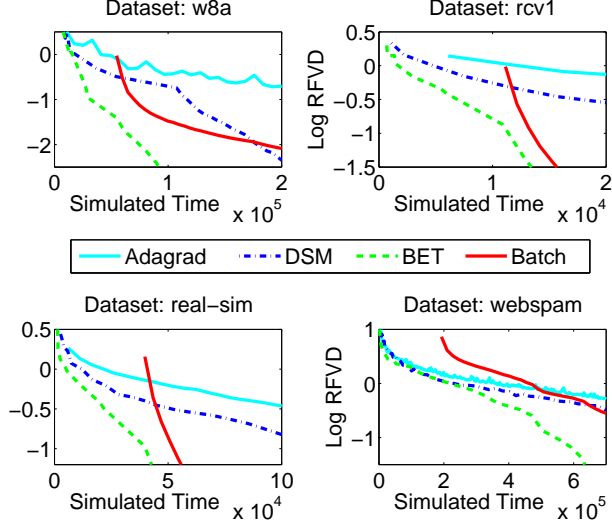


Figure 2. Log RFVD (see (6)) of the algorithms plotted against simulated runtime, with $p = 10$, $a = 1$ and $s = 5$ (see Section 4.2). BET performs better than other methods on all data sets.

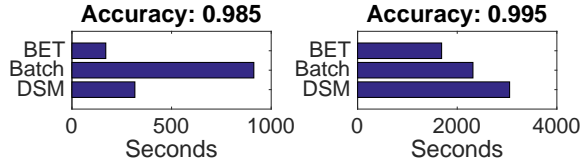


Figure 3. Wallclock times until reaching accuracy of 0.985 (within 1% of optimum) and 0.995 (within 0.05% of optimum) for the webspam dataset.

Figure 2 plots the simulated runtime with $a = 1$, $p = 10$ and $s = 5$, which is based on the system characteristics of a standard hardware setup. As expected from the time complexity analysis, DSM and Adagrad pay a significant cost for resampling the data at every iteration. We also show wallclock times until reaching test-set accuracy of 0.985 (within 1% of optimum) and 0.995 (within 0.05% of optimum) for BET, DSM and Batch, when ran on the same platform, for the webspam dataset (see Figure 3). Wallclock times for Adagrad were much larger on our platform, and so they are omitted in Figure 3. We observe that Batch has a considerable disadvantage when the chosen error tolerance is large, hence it is not suited for early stopping. BET achieves best performance for any tolerance.

To analyze our time complexity model more generally, first note that for the sake of performance comparison we can reduce it to just two degrees of freedom: $\bar{p} = a \cdot p$ and $\bar{s} = s/a$. Increasing \bar{p} means reducing computation time relative to the data availability rate. This can be caused by slow disk-loading time relative to performing operations on

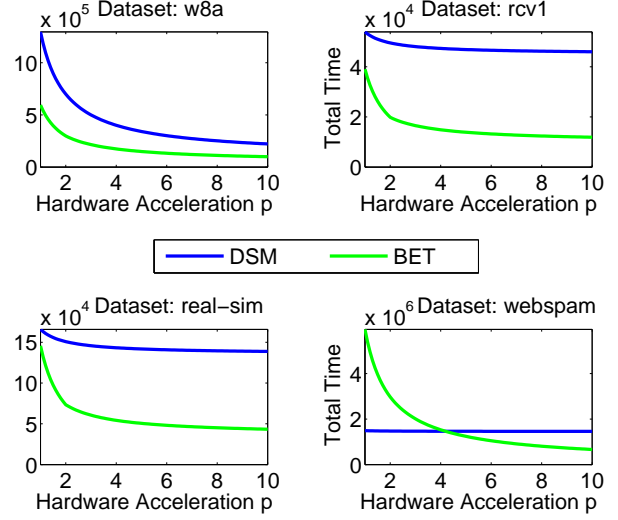


Figure 4. The effect of hardware acceleration on DSM and BET algorithms. The X axis varies parameter p with $a = 1$ and $s = 5$. Note, that due to gradual rate of data availability (controlled by a), the curve has to plateau. BET makes better use of acceleration.

the memory, as well as hardware acceleration through parallel and distributed computing. Parameter \bar{s} corresponds to the time overhead spent in between each iteration of the algorithm. We observe that:

1. Increasing \bar{p} allows BET to take advantage of running multiple updates on the same data subset (Figure 4).
2. With increasing \bar{s} , Adagrad’s relative performance degrades, since it relies on significantly more sequential updates than either BET or DSM.

5.2. Parallel Experiments

In this section, we present preliminary results for parallel experiments comparing Batch-Expansion Training with the standard Batch strategy, when using L-BFGS as the inner optimizer. Both methods were implemented in the PETSc (Balay et al., 2016a;b; 1997) framework, with the data split between multiple computing cores. Figure 5 shows the runtime results, in terms of the training objective, for optimizing the regularized logistic loss on the SUSY dataset. Dashed lines correspond to sequential experiments, whereas solid lines describe the convergence when two cores are performing computations in parallel. In this experiment, BET achieved 1.84x speed-up, whereas Batch improved 1.78x. Moreover, we can see that the advantage of BET over Batch increases when dealing with large sample sizes (4 million instances), which follows the asymptotic analysis from Section 4.2.

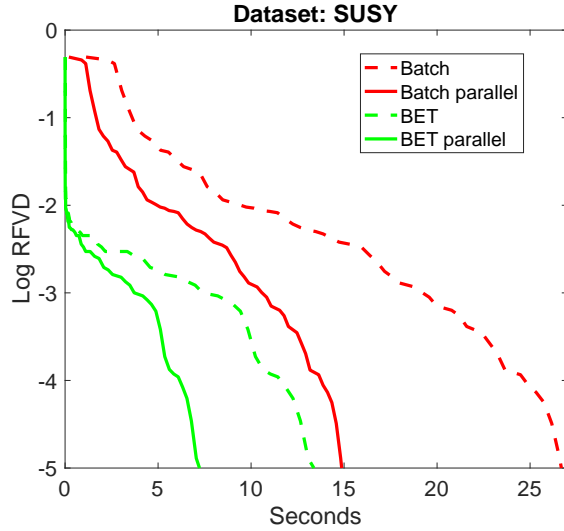


Figure 5. The effect of parallelization on Batch and BET, in terms of Log RFVD (see (6)). We compare a sequential experiment against using two computing cores in parallel. BET can take advantage of parallelization just as well as Batch.

5.3. Test Set Accuracy

Finally, we look at how the optimization performance translates to classification accuracy on the test set. Figure 6 shows test set accuracy against simulated runtime under the same time complexity model as Figure 2. We observe similar results, as BET performs better than other methods. On the plots, we marked the moment when BET reaches full dataset. In most cases, by this point the algorithm reaches close to optimum test accuracy. Thus, in many practical applications we can end the optimization by relying on this simple stopping criterion. Moreover, in problems when the full dataset is too abundant, BET will attain optimal performance well before the full data size is reached.

6. Conclusions

We proposed Batch-Expansion Training, a simple parameter-free method of running batch optimizers in a way that is both theoretically and experimentally competitive with stochastic approaches. BET does not need any tuning and can be paired with various optimizers, while offering advantages in parallel and distributed settings.

References

Agarwal, Alekh, Chapelle, Olivier, Dudík, Miroslav, and Langford, John. A reliable effective terascale linear learning system. *J. Mach. Learn. Res.*, 15(1):1111–1133, January 2014. ISSN 1532-4435. URL <http://dl.acm.org/citation>.

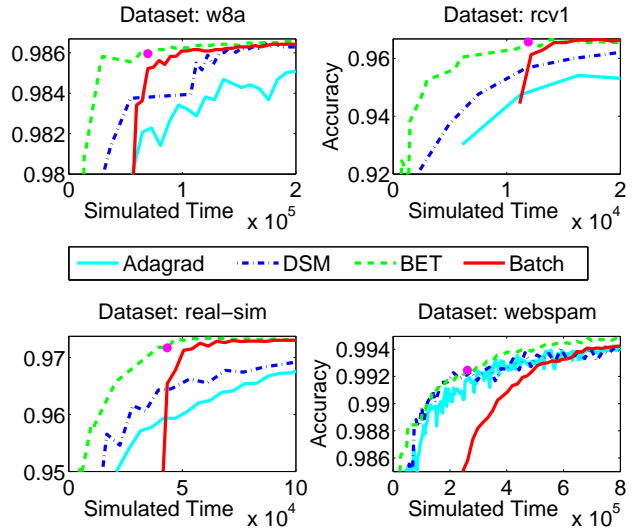


Figure 6. Test set accuracy of the algorithms plotted against simulated runtime, with $p = 10$, $a = 1$ and $s = 5$ (see Section 4.2). BET performs better than other methods on all data sets. A circular dot marks when BET reaches full dataset.

[cfm?id=2627435.2638571](https://arxiv.org/abs/1708.02702).

Authors. Batch-expansion training: An efficient optimization paradigm for machine learning (supplement). 2017.

Balay, Satish, Gropp, William D., McInnes, Lois Curfman, and Smith, Barry F. Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M., and Langtangen, H. P. (eds.), *Modern Software Tools in Scientific Computing*, pp. 163–202. Birkhäuser Press, 1997.

Balay, Satish, Abhyankar, Shrirang, Adams, Mark F., Brown, Jed, Brune, Peter, Buschelman, Kris, Dalcin, Lisandro, Eijkhout, Victor, Gropp, William D., Kaushik, Dinesh, Knepley, Matthew G., McInnes, Lois Curfman, Rupp, Karl, Smith, Barry F., Zampini, Stefano, Zhang, Hong, and Zhang, Hong. PETSc Web page. <http://www.mcs.anl.gov/petsc>, 2016a. URL <http://www.mcs.anl.gov/petsc>.

Balay, Satish, Abhyankar, Shrirang, Adams, Mark F., Brown, Jed, Brune, Peter, Buschelman, Kris, Dalcin, Lisandro, Eijkhout, Victor, Gropp, William D., Kaushik, Dinesh, Knepley, Matthew G., McInnes, Lois Curfman, Rupp, Karl, Smith, Barry F., Zampini, Stefano, Zhang, Hong, and Zhang, Hong. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016b. URL <http://www.mcs.anl.gov/petsc>.

Bottou, Leon and Bousquet, Olivier. The tradeoffs of large scale learning. In *Proceedings of the 20th In-*

- ternational Conference on Neural Information Processing Systems, NIPS'07, pp. 161–168, USA, 2007. Curran Associates Inc. ISBN 978-1-60560-352-0. URL <http://dl.acm.org/citation.cfm?id=2981562.2981583>.
- Bubeck, Sébastien. Convex optimization: Algorithms and complexity. *Found. Trends Mach. Learn.*, 8(3-4): 231–357, November 2015. ISSN 1935-8237. doi: 10.1561/22000000050. URL <http://dx.doi.org/10.1561/22000000050>.
- Byrd, Richard H., Chin, Gillian M., Neveitt, Will, and Nocedal, Jorge. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011. URL <http://dblp.uni-trier.de/db/journals/siamjo/siamjo21.html#ByrdCNN11>.
- Byrd, Richard H., Chin, Gillian M., Nocedal, Jorge, and Wu, Yuchen. Sample size selection in optimization methods for machine learning. *Math. Program.*, 134(1): 127–155, August 2012. ISSN 0025-5610. doi: 10.1007/s10107-012-0572-5. URL <http://dx.doi.org/10.1007/s10107-012-0572-5>.
- Defazio, Aaron, Bach, Francis R., and Lacoste-Julien, Simon. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 1646–1654, 2014.
- Dekel, Ofer, Gilad-Bachrach, Ran, Shamir, Ohad, and Xiao, Lin. Optimal distributed online prediction using mini-batches. *J. Mach. Learn. Res.*, 13(1):165–202, January 2012. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=2503308.2188391>.
- Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=1953048.2021068>.
- Erdogdu, Murat A. and Montanari, Andrea. Convergence rates of sub-sampled newton methods. In *Proceedings of the 28th International Conference on Neural Information Processing Systems, NIPS'15*, pp. 3052–3060, Cambridge, MA, USA, 2015. MIT Press. URL <http://dl.acm.org/citation.cfm?id=2969442.2969580>.
- Fletcher, R. and Reeves, C. M. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, February 1964. doi: 10.1093/comjnl/7.2.149. URL <http://dx.doi.org/10.1093/comjnl/7.2.149>.
- Friedlander, Michael P. and Schmidt, Mark. Hybrid deterministic-stochastic methods for data fitting. *SIAM Journal on Scientific Computing* 34(3), 2012.
- Johnson, Rie and Zhang, Tong. Accelerating stochastic gradient descent using predictive variance reduction. In Burges, Christopher J. C., Bottou, Lon, Ghahramani, Zoubin, and Weinberger, Kilian Q. (eds.), *NIPS*, pp. 315–323, 2013. URL <http://dblp.uni-trier.de/db/conf/nips/nips2013.html#Johnson013>.
- Li, Mu, Zhang, Tong, Chen, Yuqiang, and Smola, Alexander J. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14*, pp. 661–670, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623612. URL <http://doi.acm.org/10.1145/2623330.2623612>.
- Lin, Chih-Jen, Weng, Ruby C., and Keerthi, S. Sathiya. Trust region newton methods for large-scale logistic regression. In *Proceedings of the 24th International Conference on Machine Learning, ICML '07*, pp. 561–568, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-793-3. doi: 10.1145/1273496.1273567. URL <http://doi.acm.org/10.1145/1273496.1273567>.
- Matsushima, Shin, Vishwanathan, S.V.N., and Smola, Alexander J. Linear support vector machines via dual cached loops. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12*, pp. 177–185, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1462-6. doi: 10.1145/2339530.2339559. URL <http://doi.acm.org/10.1145/2339530.2339559>.
- Narayanamurthy, Shravan, Weimer, Markus, Mahajan, Dhruv, Condie, Tyson, Sellamanickam, Sundararajan, and Selvaraj, Keerthi. Towards resource-elastic machine learning, January 2013.
- Nocedal, Jorge and Wright, Steve J. *Numerical optimization*. Springer Series in Operations Research and Financial Engineering. Springer, Berlin, 2006. ISBN 978-0387-30303-1. URL <http://opac.inria.fr/record=b1120179>. NEOS guide <http://www-fp.mcs.anl.gov/otc/Guide/>.
- Schmidt, Mark W., Roux, Nicolas Le, and Bach, Francis R. Minimizing finite sums with the stochastic average gradient. *CoRR*, abs/1309.2388, 2013. URL <http://arxiv.org/abs/1309.2388>.

Shalev-Shwartz, Shai and Srebro, Nathan. Svm optimization: Inverse dependence on training set size. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pp. 928–935, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390273. URL <http://doi.acm.org/10.1145/1390156.1390273>.

Shalev-Shwartz, Shai and Zhang, Tong. Stochastic dual coordinate ascent methods for regularized loss. *J. Mach. Learn. Res.*, 14(1):567–599, February 2013. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=2502581.2502598>.

Sridharan, Karthik, Shalev-shwartz, Shai, and Srebro, Nathan. Fast rates for regularized objectives. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L. (eds.), *Advances in Neural Information Processing Systems 21*, pp. 1545–1552. Curran Associates, Inc., 2009.

Tolstikhin, Ilya, Zhivotovskiy, Nikita, and Blanchard, Gilles. Permutational rademacher complexity. In *Proceedings of the 26th International Conference on Algorithmic Learning Theory - Volume 9355*, pp. 209–223, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-319-24485-3. doi: 10.1007/978-3-319-24486-0_14. URL http://dx.doi.org/10.1007/978-3-319-24486-0_14.

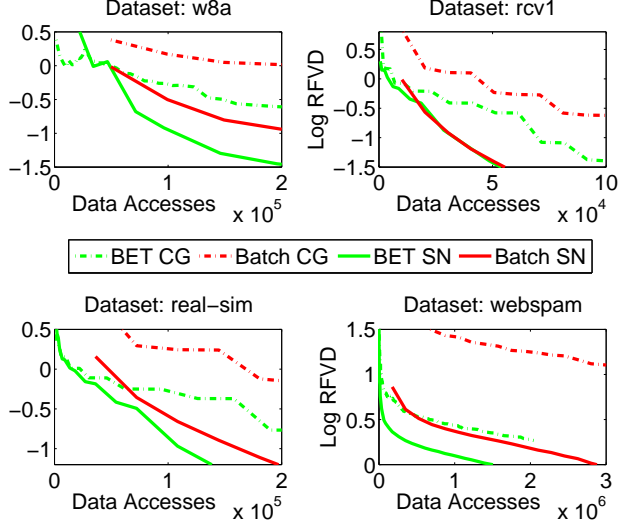


Figure 7. Log RFVD (see (6)) of BET and Batch, comparing Nonlinear-CG (CG) and sub-sampled Newton-CG (SN) as inner optimizers, plotted against the number of data accesses. BET provides significant improvement over Batch in both cases.

A. Additional Experiments

A.1. Inner Optimizers

To demonstrate the flexibility of our framework, in this section we use two different inner optimizers with BET:

1. Nonlinear Conjugate Gradient method (CG), using Fletcher-Reeves (Fletcher & Reeves, 1964) formula, with exact line-search;
2. The Sub-sampled Newton-CG (SN) algorithm, described in (Byrd et al., 2012).

Both of those methods are linear optimizers that employ strategies for enhancing the basic gradient descent direction. Note, that CG uses a memory vector updated at each iteration, which becomes invalid after every batch expansion, when the loss function changes from \hat{f}_t to \hat{f}_{t+1} . To overcome this, we restart the CG update at each stage of the training. Despite this, we show that BET can still be effective at accelerating memory-based algorithms. Sub-sampled Newton-CG does not require memory, since its updates are self-contained. Instead, it sub-samples a portion of the available data to get an estimate Hessian, and approximately find the Newton direction by performing a number of linear CG steps.

Figure 7 shows the performance comparison of using BET with the two inner optimizers, contrasted with both of them ran in regular batch mode. First, we observe that subsampled Newton-CG is a much more effective optimizer than Nonlinear CG. Moreover, both of them significantly benefit

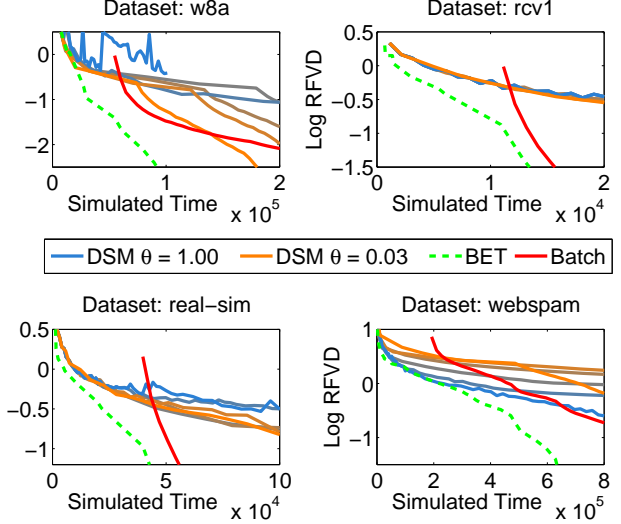


Figure 8. Performance of DSM for the following values of parameter θ : 1, 0.5, 0.2, 0.1, 0.05, 0.03 (compared with BET and Batch). Note, that for data sets w8a and real-sim clearly small values around 0.03 are needed, whereas for webspam larger values around 1 perform considerably better.

from BET. In particular, note that in the cases where data is especially poorly conditioned for CG (e.g. Webspam), using BET helps overcome this problem, putting the method in the rough range of Newton-CG performance.

Note that, similarly to BET, DSM is a general framework which can be paired with different inner optimizers, however it has an additional limitation that the optimizer cannot use memory accross updates, due to resampling. Thus, this method cannot be used with the CG update unless we restart CG at every step, turning it into plain gradient descent.

A.2. Parameter Tuning

Most stochastic optimization methods require parameter tuning to exhibit good performance. This requires restarting the algorithm many times for different parameter values, affecting the overall computation time and ease of use. Batch optimizers, on the other hand, tend to be much less sensitive to their initialization. Here, we look at how the two hybrid methods, DSM and BET, compare in this regard. Both of the approaches rely on an inner batch optimizer. The Sub-sampled Newton-CG algorithm does require us to select the fraction $R \in [0, 1]$ of data used to approximate the Hessian, and how many linear CG steps to perform on that fraction. As discussed in (Byrd et al., 2012; 2011), the method is relatively robust to those parameters, and in our experiments setting $R = 0.1$ and using $R^{-1} = 10$ steps of CG worked well on all datasets.

Turning to the hybrid meta-algorithms, as we noted earlier, BET is essentially parameter free, since it is very robust to the choice of initial subset size n_0 (which may be as small as it is practical for a given computational setting). DSM also requires setting the initial sample size n_0 , and in addition has an extra parameter θ , both of which require some tuning due to the stochastic nature of this method.

The sample size selection in DSM is based on comparing two quantities: norm of the gradient of the objective, and a variance estimate of that gradient with respect to the randomness of the sample. The intuition is that if variance is smaller than the norm, then the current sample size is sufficient to perform an effective batch update. Otherwise, we should increase the sample size so that the condition is satisfied once again. Thus, we need a parameter θ describing the variance-to-norm ratio above which we decide to increase the sample size. As shown in Figure 8, the choice of θ considerably affects the performance (and even convergence) of DSM. The value optimal for one dataset may give subpar results for another. Moreover, setting the initial sample size n_0 too small may lead to the gradient variance estimate being inadequate, which further disrupts the algorithm. Note, that BET does not suffer from any of those issues, making it much easier to use in practice.

B. Proof Details for Theorem 4.1

B.1. Proof of Theorem 4.1

Recall that we denote the approximation error estimate at stage t by $\hat{g}_t(\mathbf{w}) \triangleq \hat{f}_t(\mathbf{w}) - \hat{f}_t(\hat{\mathbf{w}}_t^*)$ and the full approximation error as $\hat{g}(\mathbf{w}) \triangleq \hat{f}(\mathbf{w}) - \hat{f}(\hat{\mathbf{w}}^*)$. Moreover, note that Algorithm 3 defines the optimization error tolerances recursively as $\epsilon_{t+1} = \epsilon_t/2$. First, we need a uniform convergence result, which is a variant of the one given in (Sridharan et al., 2009):

Lemma 1 *For any n_0, δ, T , there exists ϵ_0 such that*

$$\epsilon_0 = \mathcal{O}(L^2 B^2 \log(T/\delta) \cdot \lambda^{-1}),$$

and with probability $1 - \delta$, for all \mathbf{w} and all $0 \leq t < T$ the following hold:

$$\hat{g}_0(\mathbf{w}_0) \leq \epsilon_0, \quad (7)$$

$$\hat{g}_{t+1}(\mathbf{w}) \leq 2 \cdot \hat{g}_t(\mathbf{w}) + \epsilon_t, \quad (8)$$

$$\hat{g}(\mathbf{w}) \leq 2 \cdot \hat{g}_T(\mathbf{w}) + \epsilon_T. \quad (9)$$

See Section B.2 for proof. Next, using this lemma, we show the main result.

Proof (of Theorem 4.1) Let κ be the convergence rate of the inner optimizer. Recall, that we set the number of inner iterations of Algorithm 3 to be

$$\hat{\kappa} \triangleq \lceil \kappa \log(6) \rceil.$$

This gives us the following bound for the progress that the inner optimizer makes at each stage of the algorithm (for any $0 \leq t \leq T$):

$$\begin{aligned} \hat{g}_{t+1}(\mathbf{w}_{t+1}) &\leq \left(1 - \frac{1}{\kappa}\right)^{\hat{\kappa}} \hat{g}_{t+1}(\mathbf{w}_t) \\ &\leq \exp\left(-\frac{\hat{\kappa}}{\kappa}\right) \hat{g}_{t+1}(\mathbf{w}_t) \\ &\leq \frac{\hat{g}_{t+1}(\mathbf{w}_t)}{6}. \end{aligned}$$

Suppose that ϵ_0 satisfies Lemma 1. We can show by induction that (with probability $1 - \delta$) for all $t \leq T$, model \mathbf{w}_t is an ϵ_t -approximate solution for \hat{f}_t , i.e. that $\hat{g}_t(\mathbf{w}_t) \leq \epsilon_t$. Base case is given by (7). The inductive step follows from:

$$\begin{aligned} \hat{g}_{t+1}(\mathbf{w}_{t+1}) &\leq \frac{\hat{g}_{t+1}(\mathbf{w}_t)}{6} \leq \frac{2 \cdot \hat{g}_t(\mathbf{w}_t) + \epsilon_t}{6} \\ &\leq \frac{2\epsilon_t + \epsilon_t}{6} = \frac{\epsilon_t}{2} = \epsilon_{t+1}. \end{aligned}$$

Next, we verify that \mathbf{w}_T is an ϵ -approximate solution for \hat{f} :

$$\hat{g}(\mathbf{w}_T) \leq 2 \cdot \hat{g}_T(\mathbf{w}_T) + \epsilon_T \leq 3 \cdot \epsilon_T \leq \epsilon.$$

Finally, we move on to complexity analysis. The number of iterations in the algorithm is $T = \mathcal{O}(\log(\epsilon_0/\epsilon))$ as shown by:

$$2^T = \frac{2 \cdot \epsilon_0}{\epsilon_{T-1}} < 6 \cdot \frac{\epsilon_0}{\epsilon}.$$

Assuming that one update of the inner optimizer requires C passes over the data, we obtain the data-access complexity:

$$\begin{aligned} \sum_{t=1}^T \hat{\kappa} C n_t &= C \hat{\kappa} n_0 \sum_{t=1}^T 2^t \\ &\leq 2C \hat{\kappa} n_0 \cdot 2^T = \mathcal{O}\left(n_0 \epsilon_0 \cdot \frac{\kappa}{\epsilon}\right). \end{aligned}$$

See Section B.3 for details regarding the log terms. \square

B.2. Proof of Lemma 1

First, we formulate a uniform-convergence bound, which closely resembles Theorem 1 from (Sridharan et al., 2009). The only difference is that they consider PAC setting: sampling an i.i.d. dataset \mathbf{Z} from a fixed distribution, and comparing the finite-sample objective \hat{f} computed using \mathbf{Z} with the true objective f , which is an expectation over the distribution. On the other hand, we consider an increasing sequence of datasets $\mathbf{Z}_t = \{z_i\}_{i=1}^{n_t}$, selected by a uniformly random permutation of the full dataset \mathbf{Z} . Note, that we assume the algorithm never observes the full dataset, only loading as much data as needed. Taking the limit of

$N \rightarrow \infty$, the relationship between any subset \mathbf{Z}_t and the full dataset Z becomes statistically equivalent to i.i.d. sampling from any fixed underlying distribution. Given that our goal is generalization to predicting on new data, that simplification is reasonable, although the analysis does go through in the strict optimization setting, where N is finite. However, even with this assumption, we still need to describe the relationship between two consecutive subsets in the sequence, which does not fit the i.i.d. sampling model. To that end, we can view \mathbf{Z}_t as a fraction of elements from \mathbf{Z}_{t+1} , selected uniformly at random without replacement. We now describe the relationship between the two consecutive loss estimates in this sequence. Note, that in this section the big- \mathcal{O} notation hides only fixed numeric constants.

Lemma 2 *With probability $1 - \delta$, for all \mathbf{w} and all $0 \leq t \leq T$ we have*

$$\hat{g}_{t+1}(\mathbf{w}) \leq 2\hat{g}_t(\mathbf{w}) + \mathcal{O}\left(\frac{L^2 B^2 \log(T/\delta)}{\lambda n_t}\right). \quad (10)$$

Proof The proof is very similar to (Sridharan et al., 2009), except we replace standard Rademacher Complexity with Permutational Rademacher Complexity (PRC), proposed in (Tolstikhin et al., 2015). Let us fix t , and consider a specific set of instances \mathbf{Z}_{t+1} , from which a random subset \mathbf{Z}_t is sampled (without replacement). Following (Sridharan et al., 2009), for any $r > 0$ we define

$$\mathcal{H}_{t,r} \triangleq \left\{ h_{\mathbf{w}}^{t,r} = \frac{h_{\mathbf{w}}^t}{4^{k_{t,r}(\mathbf{w})}} : \mathbf{w} \in \mathbf{W} \right\},$$

where

$$k_{t,r}(\mathbf{w}) \triangleq \min\{k' \in \mathbb{Z}_+ : \hat{f}_{t+1}(\mathbf{w}) \leq r 4^{k'}\}$$

and

$$h_{\mathbf{w}}^t(z) \triangleq \ell_z(\mathbf{w}) - \ell_z(\hat{\mathbf{w}}_{t+1}^*).$$

Our aim is to analyze the empirical average of the function values from $\mathcal{H}_{t,r}$ evaluated on a given instance set \mathbf{Z} :

$$\bar{h}_{\mathbf{Z}}^{t,r}(\mathbf{w}) = \frac{1}{|\mathbf{Z}|} \sum_{z \in \mathbf{Z}} h_{\mathbf{w}}^{t,r}(z).$$

We can translate the task of comparing \hat{g}_{t+1} and \hat{g}_t to describe it in terms of the function class $\mathcal{H}_{t,r}$:

$$\begin{aligned} \hat{g}_{t+1}(\mathbf{w}) - \hat{g}_t(\mathbf{w}) &= \hat{g}_{t+1}(\mathbf{w}) - (\hat{f}_t(\mathbf{w}) - \hat{f}_t(\hat{\mathbf{w}}_t^*)) \\ &\leq \hat{g}_{t+1}(\mathbf{w}) - (\hat{f}_t(\mathbf{w}) - \hat{f}_t(\hat{\mathbf{w}}_{t+1}^*)) \\ &= 4^{k_{t,r}(\mathbf{w})} \left[\bar{h}_{\mathbf{Z}_{t+1}}^{t,r}(\mathbf{w}) - \bar{h}_{\mathbf{Z}_t}^{t,r}(\mathbf{w}) \right]. \end{aligned}$$

To compare $\bar{h}_{\mathbf{Z}_t}^{t,r}$ with $\bar{h}_{\mathbf{Z}_{t+1}}^{t,r}$ we use Theorem 5 (Tolstikhin et al., 2015), which provides transductive risk bounds

through expected PRC of function class $\mathcal{H}_{t,r}$, conditioned on set \mathbf{Z}_{t+1} :

$$\mathcal{Q}(\mathcal{H}_{t,r}, \mathbf{Z}_{t+1}) \triangleq \mathbb{E} \left[\hat{Q}_{n_t, n_t/2}(\mathcal{H}_{t,r}, \mathbf{Z}_t) \mid \mathbf{Z}_{t+1} \right].$$

Here, the randomness only comes from selecting \mathbf{Z}_t as a subset of \mathbf{Z}_{t+1} . For any $\delta > 0$, with probability at least $1 - \delta$,

$$\begin{aligned} &\sup_{\mathbf{w}} \left[\bar{h}_{\mathbf{Z}_{t+1}}^{t,r}(\mathbf{w}) - \bar{h}_{\mathbf{Z}_t}^{t,r}(\mathbf{w}) \right] \\ &\leq \underbrace{\mathcal{Q}(\mathcal{H}_{t,r}, \mathbf{Z}_{t+1})}_{Y_1} + \underbrace{\sup_{h_{\mathbf{w}}^{t,r}, z} |h_{\mathbf{w}}^{t,r}(z)| \cdot \mathcal{O}\left(\sqrt{\frac{\log(1/\delta)}{n_t}}\right)}_{Y_2}. \end{aligned}$$

Note, that $\mathcal{Q}(\mathcal{H}_{t,r}, \mathbf{Z}_{t+1}) = \mathcal{O}(\mathcal{R}_{n_t}(\mathcal{H}_{t,r}))$ (see (Tolstikhin et al., 2015)), where \mathcal{R}_{n_t} is the standard Rademacher Complexity. The remainder of the proof proceeds identically as in (Sridharan et al., 2009) (up to numerical constants), i.e. by bounding both terms Y_1 and Y_2 by

$$\mathcal{O}\left(LB\sqrt{\frac{r \log(1/\delta)}{\lambda n_t}}\right).$$

Note, that since the bound is obtained for every possible \mathbf{Z}_{t+1} , it will still hold with probability at least $1 - \delta$ without conditioning on \mathbf{Z}_{t+1} .

Finally, as shown in (Sridharan et al., 2009), by setting r appropriately we obtain that w.p. $1 - \delta$, for all \mathbf{w}

$$\hat{g}_{t+1}(\mathbf{w}) \leq 2\hat{g}_t(\mathbf{w}) + \mathcal{O}\left(\frac{L^2 B^2 \log(1/\delta)}{\lambda n_t}\right).$$

Applying union bound to account for all values of t simultaneously, we obtain the desired result. \square

We return to the proof of Lemma 1. Using Lipschitz and boundedness assumptions for the loss ℓ and mapping ϕ , as well as strong convexity of the regularized objective, we obtain initial tolerance of the loss estimate:

$$\begin{aligned} \hat{g}_0(\mathbf{w}_0) &= \hat{f}_0(\mathbf{w}_0) - \hat{f}_0(\hat{\mathbf{w}}_0^*) \\ &\leq \frac{1}{n_0} \sum_{i=1}^{n_0} (\ell_{z_i}(\mathbf{w}_0) - \ell_{z_i}(\hat{\mathbf{w}}_0^*)) \\ &\leq LB \|\hat{\mathbf{w}}_0^*\| \leq LB \sqrt{\frac{2\hat{g}_0(\mathbf{w}_0)}{\lambda}}, \\ \hat{g}_0(\mathbf{w}_0) &\leq \frac{2L^2 B^2}{\lambda}. \end{aligned}$$

We used the fact that \mathbf{w}_0 is set to zero only for applying inequality $\|\mathbf{w}_0\| \leq \|\hat{\mathbf{w}}_0^*\|$ to drop the regularization terms (hence, any initialization satisfying that requirement is acceptable).

Finally, Condition (9) regards the relationship between approximation error estimate \hat{g}_T and full approximation error \hat{g} . This bound can be obtained by repeating the same argument as in Lemma 2. We can either assume $N \rightarrow \infty$ and use standard Rademacher complexity, as in Theorem 1, (Sridharan et al., 2009), or stay with the finite optimization model and apply PRC. Thus, we can clearly set ϵ_0 to satisfy the conditions of Lemma 1. \square

B.3. Deriving Log Terms in Theorem 4.1

The number of iterations, $T = \mathcal{O}(\log(\epsilon_0/\epsilon))$, depends on ϵ_0 . But in Lemma 1 we defined ϵ_0 using T . To address this, we have to find ϵ_0 satisfying:

$$\epsilon_0 \geq K \log \left(\frac{\log(\epsilon_0/\epsilon)}{\delta} \right),$$

with $K = \mathcal{O}(L^2 B^2 / \lambda)$. It is easy to show that for small enough ϵ it suffices to set

$$\begin{aligned} \epsilon_0 &\triangleq 2K \log \left(\frac{\log(1/\epsilon)}{\delta} \right) \\ &= \mathcal{O} \left(\frac{L^2 B^2}{\lambda} \cdot (\log \log(1/\epsilon) + \log(1/\delta)) \right). \end{aligned}$$

Thus, setting $n_0 = 1$, we obtain the final complexity bound in Theorem 4.1 as

$$\mathcal{O} \left(\frac{\kappa}{\lambda \epsilon} \cdot L^2 B^2 \cdot (\log \log(1/\epsilon) + \log(1/\delta)) \right).$$